by
Ray Duncan

Power Programming

Containers and Objects: Object Linking and Embedding in Windows

- **A new protocol for the construction of compound documents, OLE is built upon DDE but represents the relationships among document components at a much higher level.**

Although you might think *Microsoft Windows* would be an ideal environment for complex suites of interdependent applications, given its multitasking and virtual memory capabilities, the interprocess-communications (IPC) facilities in *Windows* have been slow in evolving. In fact, *Windows* 1.0 had no standardized IPC facilities other than the system clipboard, and the dynamic data exchange (DDE) protocol that was defined in *Windows* 2.0 was cobbled together out of global atoms, global data segments, and a few special-purpose *Windows* messages. As we saw in the last two installments of this column, DDE is certainly better than nothing, but it is a fragile and complex mechanism, and it requires skill, experience, and a generous measure of good luck on the part of the programmer.

Nevertheless, you might be surprised to learn that *Windows'* track record for IPC, when compared with that of a far more elegant GUI environment—the Macintosh System—is really quite good. Even today, in Macintosh System 6.0.7, the only provision for transferring data from one application to another is the clipboard, and the clipboard would be extremely painful to use if not for the relatively recent addition of a simple task switcher (MultiFinder) that allows you to keep more than one program in memory at once. There is no Mac equivalent to a DDE-based hot link; all transports of data are passive and static, relying entirely on user intervention.

In 1989 and 1990, significant pressure was placed on Microsoft to make *Windows* into a better platform for the synergistic use of multiple applications. This pressure came from two directions. First, Hewlett-Packard released *HP NewWave Environment,* an object-oriented user and application interface that is layered on top of *Windows*. *NewWave* allows text and graphics produced by multiple applications to be combined in a natural manner. It also introduces the concept of *agents*—the first step toward user interfaces that "do what I mean, not what I say." Although the critics loved *NewWave,* it has been slow to find acceptance among users; this results partially from its price, partially from its demand for system resources, and partially (I suspect) from its status as a proprietary product of a PC hardware vendor. *PC Magazine* has reviewed *NewWave* in detail, so I won't discuss it further here.

Second, during 1989, Apple released detailed information about System 7, the upcoming major overhaul of the Macintosh's operating system, and seeded its key developers with beta-test versions of the software in 1990. The capabilities of System 7 stand in approximately the same relationship to System 6 (the current version) as the capabilities of OS/2 do to DOS; we can only hope for the sake of Macintosh users that System 7 meets a kinder fate than OS/2 did! In any case, among the new features of System 7 is a multilevel IPC facility called the Apple Inter-Applications Architecture (IAC), which has four facets: program-to-program communication, Clipboard Copy-Paste, Live Copy-Paste, and AppleEvents.

Program-to-program communication

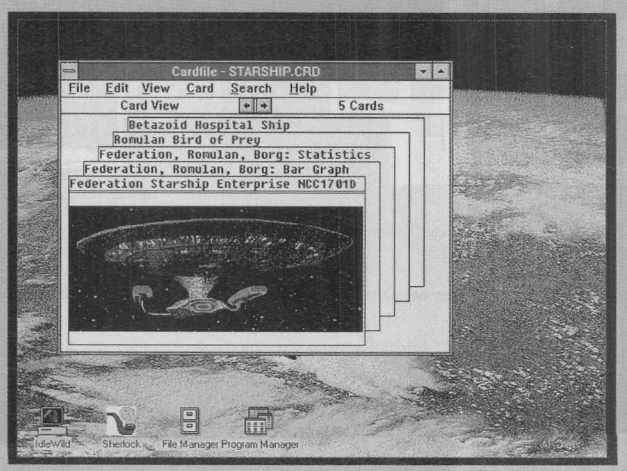

Figure 1: The first step in editing an embedded object. This shows an experimental OLE-aware version of Cardfile, which has loaded a Cardfile file containing several different embedded objects. The card at the front of the stack contains a picture that was pasted in from *Paintbrush*.

in System 7 is roughly equivalent to the semaphores, pipes, and shared memory segments found in OS/2 or Unix; it allows raw data to be passed from one program to another but gives the programs no aid in interpreting the data. One important difference between System 7's program-to-program communication and the more traditional Unix or OS/2 mechanisms is that System 7 will support both real-time data transfer and store-and-forward communication. Clipboard Copy-Paste is the mechanism for transfer of text and bitmaps that has been available on the Mac from the beginning; it is essentially a passive operation from the application's point of view, but it does aid the application by imposing structure on the data.

System 7's Live Copy-Paste is conceptually similar to the DDE support in *Windows* and OS/2. It allows the data in worksheets, databases, documents, and other application constructs to be hot-linked so that a change made to one file will automatically be reflected in the others. The last of the new IAC facilities, AppleEvents, has the most long-term potential. AppleEvents are a standard set of messages that applications can use to request actions of one another, mimicking in some ways the messages sent to an application when the user interacts with it via the screen and mouse. Among other things, AppleEvents will form the basis of a high-level macro language, allowing users to automate complex series of actions involving multiple application programs in an object-oriented manner.

Microsoft's own near-term plans for improving application integration under *Windows* are based on a protocol for the construction of *compound documents*—documents whose components are created in more than one application—called Object Linking and Embedding (OLE, sometimes pronounced like the Spanish word *Olé*). OLE was developed by Microsoft in collaboration with Lotus, Aldus, WordPerfect Corp., and others, and some degree of support for OLE is already present in Microsoft's latest *Windows* applications, *PowerPoint* and *Excel* 3.0. In this installment, we'll discuss OLE from several different perspectives: the user interface, the conceptual framework, and its implementation in *Windows*.

OLE AND THE USER

The capabilities of OLE are presented to the user via the standard *Windows* Copy, Cut, Paste, and Paste-Link operations. The user doesn't have to learn any new commands, and the behavior of these operations remains unchanged across non-OLE applications, or between non-OLE and OLE applications. However, when data is Copied (or Cut) in one OLE-aware application and then Pasted or Paste-Linked into another OLE-aware application, the user finds that his working environment has suddenly become a lot richer. Let's look at the Paste and Paste-Link cases individually, using *Excel* 3.0 and the OLE-modified versions of Paintbrush and Cardfile that were distributed at the recent Software Development '91 conference.

Pasted data actually resides in the document or other file into which it has been pasted; the original Copied data is physically distinct from the Pasted data, and when you change the data that was Copied, the Pasted data will not be automatically updated to match it. The new feature that OLE brings to the party is the ability of the Pasted data to "remember" which application created it, and to transparently invoke that application for editing and presentation of the Pasted data. Figure 1 shows a small Cardfile document containing different text chunks,

With OLE, pasted data has the ability to "remember" which application created it.

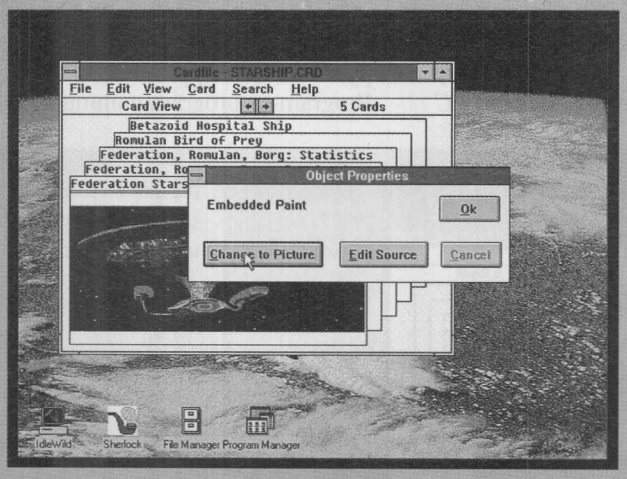

Figure 2: In order to edit the embedded picture, you first click on the picture to select it, then choose the Properties item on Cardfile's Edit menu, which results in the dialog box shown here. To invoke the application that owns the embedded object, you click on the Edit Source button.

Figure 3: The OLE libraries looked up the class information for the embedded object in the system registry, determined that this particular object was created by Paintbrush, and loaded Paintbrush as an OLE "slave" to edit the picture.

graphs, tables, and images on separate cards. The top card contains a bitmapped picture of a famous starship that was Pasted in from Paintbrush. If I click on the bitmap within the card and then select Properties from Cardfile's Edit menu, I am presented with a dialog box that gives me several options (see Figure 2). I can dismiss the dialog without doing anything (the OK button), I can throw away the OLE information associated with the bitmap, transforming the bitmap into static data within the Cardfile forever (the Change to Picture button), or I can invoke the application that knows how to understand and manipulate the bitmap (the Edit Source button).

Let's assume that I click on the Edit Source button. Cardfile continues running, but Paintbrush is loaded and becomes the active application, displaying a copy of the same bitmapped picture in its editing window (Figure 3). (Note Paintbrush's title bar, which indicates that it is running as a "slave" of another OLE application.) I am now free to make any changes that I like to the bitmap, using the usual Paintbrush tools, but the bitmap displayed in the Cardfile window doesn't reflect these changes as I make them. When it's time to leave Paintbrush, I pull down the File menu as usual and find some unfamiliar possibilities there (Figure 4): Update, Save Copy As, and Exit and Return. If I select Update, the (possibly modified) bitmap is stored back into the Cardfile card whence it came, overwriting the previous bitmap; the Save Copy As command allows me to write the bitmap into a file of its own, without any effect on the Cardfile card's bitmap; and the Exit and Return menu item closes Paintbrush down, making Cardfile the active application once again. Of course, I can also switch back and forth between Cardfile and Paintbrush at any time while both are running, by clicking on the appropriate window or by bringing up the Program Manager's Task List.

OLE Paste-Linked data, unlike Pasted data, is conceived of as existing in just one place in the system—the document or other file within which it was originally created. The document that the data is Paste-Linked into contains a bundle of pointers at the point of the data's insertion: the name of the application that created the data, the name of the file containing the data, and some coordinates or other values that define and delimit the Paste-Linked data within the original file. Before OLE, you could only modify Paste-Linked data if you were in possession of the appropriate "magical" information: you had to explicitly load the application that created the data, open the file that contained the data, make your changes, and then the file where the data was Paste-Linked would also be updated via DDE to reflect those changes. When applications are OLE-aware, nearly all of this hassle disappears.

Let's look at a real example of OLE Paste-Linking. Using the same Cardfile file, I have now brought a card to the front that contains a small table from an *Excel* 3.0 spreadsheet (Figure 5). When I click on the table and then invoke the Edit Properties command, I am presented with a much different dialog box (Figure 6) from the one we previously saw for Pasted data: This dialog allows me to choose between working on the original data (the Open Source button), forcing an update of the Cardfile card from the original document (the Update Now button), converting the Paste-Linked data into static data within the Cardfile card (the Cancel Link button), updating the Paste-Link information to reflect a change in the original file's name or location (the Change Link button), or dismissing the dialog (the Close button).

For this example, I click on the Open Source button. *Excel* is activated, and it automatically loads the spreadsheet from which the data was Paste-Linked into Cardfile. Any changes that I make in the spreadsheet window are immediately visible in the Cardfile window as well (Figure 7). This makes sense, because our mental model of Paste-Linked data is that there is really only one copy of it around. When I pull down the File menu of *Excel,* I

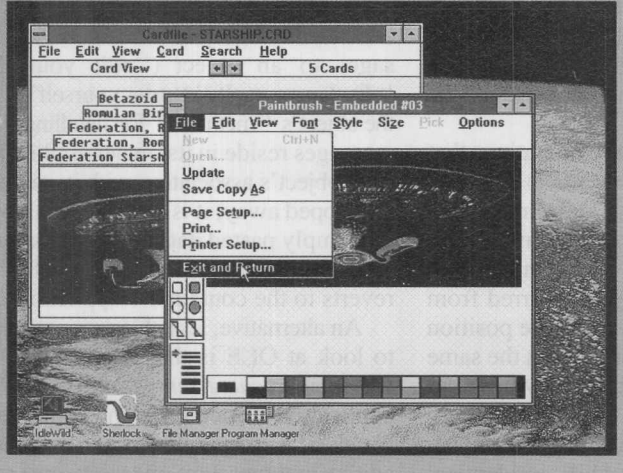

Figure 4: When Paintbrush is loaded to edit an embedded or linked object, it is passed a special command line switch that causes it to change its File menu and to respond to commands that are passed via the OLE libraries, as well as to direct interactions with the user.

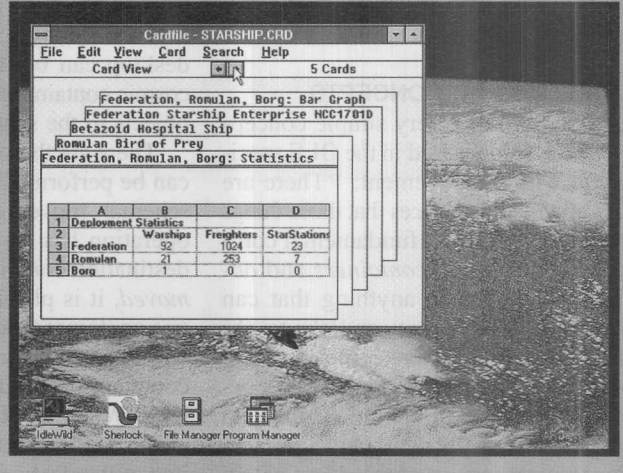

Figure 5: The first step in editing a linked object. Using the same OLE-aware version of Cardfile and the same demonstration file containing embedded and linked objects, we have now brought a card to the front that contains a table that was Paste-Linked in from an *Excel* 3.0 spreadsheet.

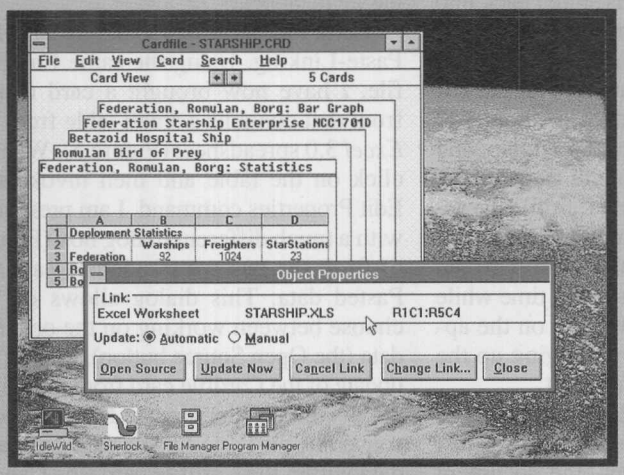

Figure 6: After clicking on the linked table and invoking the Properties item in Cardfile's Edit menu, you see a very different dialog box than the one used for an embedded object. This dialog box lets you choose between working on the original data, forcing an update of the Cardfile card, converting the Paste-Linked data into static data, updating the Paste-Linked information, or dismissing the dialog. To work on the data in the table, you click on the Open Source button.

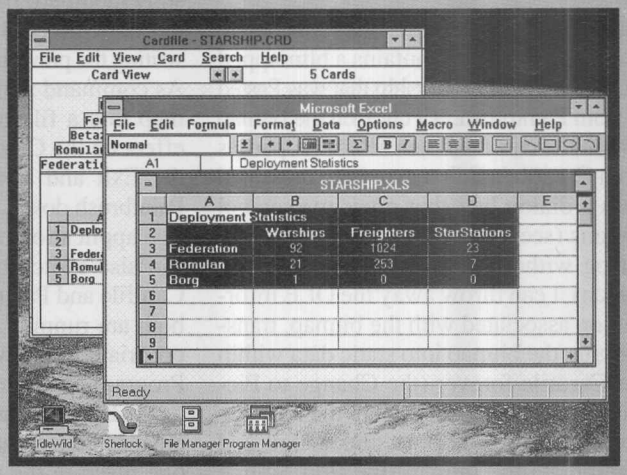

Figure 7: The OLE libraries have activated *Excel,* which has in turn loaded the original spreadsheet for editing. Changes made to the spreadsheet are reflected immediately in the Cardfile window as well. However, if *Excel* is exited without saving the modified spreadsheet, the table in the Cardfile card will revert to its previous state. This is because the data exists in only one place, and both the Cardfile and *Excel* presentations are simply different viewports onto the same data.

don't find anything new or OLE-related—just the usual Save, Save As, and Exit commands. If I save the changes and exit from *Excel,* Cardfile comes to the foreground again, and the card containing the spreadsheet data is visible with the updated information; if I terminate *Excel* or close the spreadsheet without saving the changes, the data in the Cardfile card reverts to the previous values.

OLE TERMS AND CONCEPTS

OLE is based on a very simple conceptual model, summarized in the OLE specification with the statement: ''There are things and there are places that those things can reside.'' The two fundamental components of OLE are *containers* and *objects.* A container is anything that can receive and hold objects or links to objects. An object is a more complex entity: it has *content* (that is, unique data), *behavior* (a valid set of operations that can be performed on the data), and a *presentation* (the appearance of the object within its container). The *owner* of an object is the application that created it, regardless of where the object may end up.

The prototype of a container is a file—a named collection of bytes on a disk—but the concept of a container can be taken much further. It's quite reasonable, for example, to regard a Cardfile file as a nested set of containers: The file contains a collection of cards, each of which can be thought of as an object, but a single card is also a container, because it can hold objects created in and native to Cardfile (the card's title) and objects pasted in from somewhere else (text or bitmaps in the card's body). Similarly, the *Windows* desktop can be thought of as a sort of cosmic container that encompasses all the objects in the system.

There are three atomic operations that can be performed on objects in the OLE schema: move, copy, and link. Each operation has a source container and a destination container. When an object is *moved,* it is physically transferred from one container to another (or one position within a container to another in the same container) without leaving anything behind. When an object is *copied,* the object is duplicated and one object is deposited at the destination while an identical object remains behind at the source. When an object is *linked,* it remains unchanged at the source location, but information describing the source and owner of the object is placed at the destination.

The critical aspect of compound documents (containers) built up of objects or links to objects is that the application controlling the container doesn't need to understand the objects. When an embedded or linked object is manipulated in any way other than to simply move it around, it is operated on by the tools in the application that originally created it—not by the container's application. In effect, a container just sends generic messages to an object (''edit yourself,'' ''display yourself,'' ''print yourself''), and the object's ''methods'' for handling these messages reside in its owner application. If an object's association with its methods is stripped away, it is no longer an object but simply pasted data, and the responsibility for editing and presenting the data reverts to the container's application.

An alternative, less object-oriented way to look at OLE is that it is yet another variation on *client-server* computing. The application that controls the container is the client, and it requests various kinds of editing and presentation (rendering) services from the servers associated with the objects in the container.

THE PROGRAMMER'S VIEW OF OLE

OLE is realized in *Windows* using the systemwide registration facility, several

CLIENT ECD FUNCTIONS

Document management functions

EcdRegisterClientDoc
EcdReleaseClientDoc
EcdRenameClientDoc
EcdRevertClientDoc
EcdSavedClientDoc

Clipboard handling functions

EcdCopyToClipboard
EcdCreateFromClip
EcdCreateLinkFromClip
EcdCutToClipboard
EcdQueryCreateFromClip
EcdQueryLinkFromClip

Object file I/O functions

EcdLoadFromStream
EcdSaveFromStream

Object creation functions

EcdClone
EcdCopyFromLink
EcdCreate
EcdCreateFromFile
EcdCreateFromTemplate
EcdCreateLinkFromFile
EcdEqual
EcdObjectConvert

Object management functions

EcdClose
EcdDelete
EcdDraw
EcdEnumObjectFormats
EcdGetData
EcdGetLinkUpdateOptions
EcdOpen
EcdQueryBounds
EcdQueryOpen
EcdQueryOutOfDate

EcdQueryProtocol
EcdQueryReleaseMethod
EcdQueryReleaseStatus
EcdReconnect
EcdRelease
EcdSetBounds
EcdSetData
EcdSetHostNames
EcdSetLinkUpdateOptions
EcdSetTargetDevice
EcdUpdate

Figure 8: The Client ECD functions defined by Microsoft for Object Linking and Embedding. The entry points to these functions are found in a client DLL.

new clipboard data formats, and a new API that supports and defines the communication between clients and servers.

When an OLE application is installed, it makes its presence known to other OLE applications via the generic *Windows* tration facility. (This facility is a small indexed database and a set of functions that allow information to be added, queried, or deleted from the database.) There are four crucial pieces of information placed in the system registry by each application that is capable of acting as an OLE server: the class name for the objects that it knows how to handle (often derived from the name of the application), a human-readable and more descriptive version of the class name, the exact name of the application's executable file, and the name of the application's OLE server DLL (more about this presently).

The new clipboard data formats allow an OLE-aware application to communicate objects to other OLE applications and still support the simpler, traditional Copy/Cut/Paste/Paste-Link operations as well. When the Copy command is used in an OLE application, the application first renders the data to the clipboard in the standard text, bitmap, and metafile formats that can be understood by "old" *Windows* programs. It then adds three more clipboard data formats that completely define the object. The first format is called *Native;* it is simply the object's data in the owner application's internal representation. The second format is called *OwnerLink;* this format contains the object's class name, which completely defines the object's behavior, because it can be used to look up the name of the server application for the object in the system registry. The third format is called *ObjectLink;* it describes the class of the object's source container, the filename of the container, and the limits of the object within the source container.

The API functions defined as the basis of OLE are numerous and complex. Each function name starts with Ecd (for *extensible compound document*), and the functions fall into two groups: functions called only by clients and functions called only by servers (Figures 8 and 9). The entry points to all the functions are located in dynamic link libraries (DLLs). As you can see from Figure 10, the code that does the work of OLE is distributed among five places: the client application, a client OLE DLL, a generic object management DLL, a server DLL, and the server application. All of these pieces must work in perfect harmony for OLE to function as the user expects. It's important to note that the OLE API functions are not part of *Windows* itself; the interface is defined by Microsoft, but the implementation is performed on a program-by-program basis by application vendors.

Let's walk through a highly simplified OLE scenario. When an OLE-aware client

SERVER ECD FUNCTIONS

EcdBlockServer
EcdRegisterDocument
EcdRegisterServer

EcdRevokeDocument
EcdRevokeObject
EcdRevokeServer
EcdUnblockServer

Figure 9: The entry points to these functions are found in a server DLL. As part of the information passed to the DLL during registration, the server application provides addresses of entry points for routines that implement each of the operations (edit, create, destroy, show, and so on) that can be requested by a client on an object owned by the server.

application opens a document and finds that the document contains embedded or linked objects, it calls EcdRegisterClientDocument. During loading of the file, whenever the client reaches an embedded object, it calls the function EcdLoadFromStream, which in turn calls the server library for the object's class to bring the object into memory (the server understands the format of the object, whereas the client cannot). To display the object, the client sets up a device context and bounding rectangle and calls EcdDraw, which in turn calls the rendering functions in the class-specific server library. I'm glossing over a lot of the fine details here, but I think you'll get the idea.

When the user selects an object and indicates that he wants to edit it, the client application calls EcdOpen, which looks up the filename of the server application for the object's class in the system registry and then launches the application. The server is signaled that it is running as an OLE captive by the special command line switch `-Embedding`. The server then modifies its menu commands appropriately and registers itself by calling EcdRegisterServer, passing a structure that contains the addresses of the entry points for the various document and object "methods" (open, create, create from template, edit, save, release, and so on) that the server supports. Subsequent execution of the server is controlled by commands passed through from the client or by the user's direct interaction with the server. Typically, the server will be "silent" until the client sends it a pointer to an object and

Computers, FCC Class A, Class B, and You — or When is it better to get a B than an A?

You need to know the difference between computers that meet the FCC class B radio frequency emissions standards and those that meet only the Class A standards.

Computers emit radio signals in their operation. Because these signals may cause interference to radio and television reception, the marketing and the use of computers is regulated by the Federal Communications Commission. Under federal rules, computer users are responsible for remedying interference, including interference in neighboring homes.

Computers certified by the FCC as meeting the Class B standard are less likely to cause interference to radio and TV reception than those that have been verified by the manufacturer or importer to the Class A standards. Only Class B certified computers may be advertised, sold, or leased for use in residences. A similar regulatory program applies in Canada.

Buyers seeking computers for use in homes (including offices at home) should shop for computers and peripherals which have been Class B certified. These devices carry a label with an FCC ID number. Both new and used Class A verified devices may be sold only for use in commercial and industrial locations. Signals from computers are more likely to be masked by electrical noise from other equipment in such an environment. These areas are also likely to have fewer radios and TVs. Accordingly, equipment marketed only for use in these locations may meet the less rigorous Class A standard. Class B certified equipment may be marketed for use in residences as well as commercial and industrial locations.

As you shop for a computer for use in your home, look for the FCC classification in the specifications or ask your vendor to recommend only machines that have been certified to the Class B limits. TV viewers and radio listeners in your home and in neighboring homes will be glad you did.

a command to open a window and prepare to edit the object. When the server exits, it must store away the updated object on behalf of the client and then call EcdRevokeServer to notify the OLE DLLs that its "method" entry points are no longer valid.

PROS AND CONS OF OLE

Clearly, OLE is a major step forward from the cold links, warm links, and hot links supported by today's DDE-capable *Windows* applications. OLE allows the user to deal with compound documents in a much more natural way, because the burden of finding and loading the application responsible for each object is shifted to the OLE system as a whole. As a wide variety of server class libraries becomes available, the job of client applications will become easier; applications will no longer be obligated to understand and convert or render a host of alien data formats.

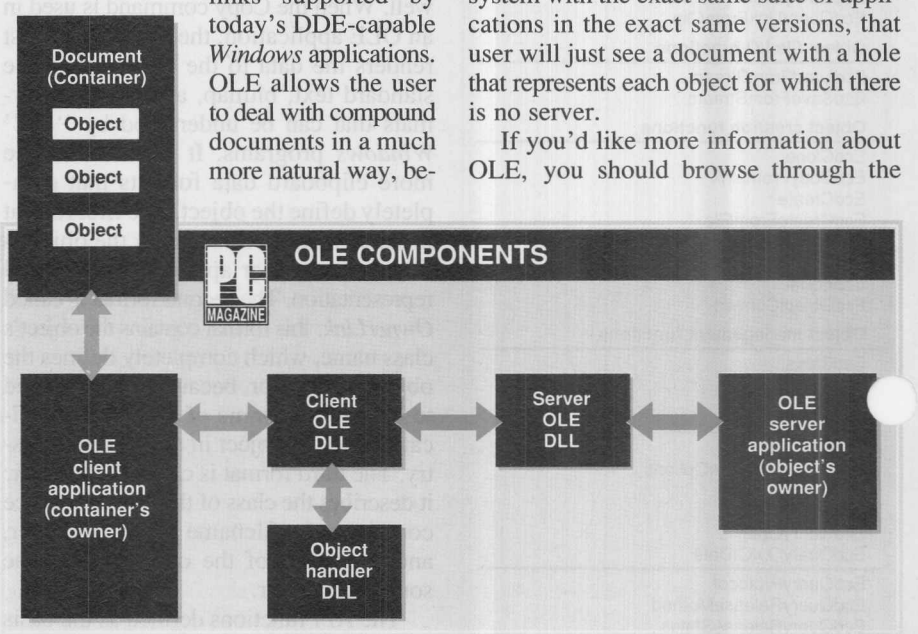

Figure 10: This illustrates the relationships among an OLE client, an OLE server, a complex document, and the DLLs that support the OLE services and API.

But there are also some storm clouds on the horizon. As currently specified, OLE relies to a significant extent on DDE to pass things around. Although the encapsulation of DDE into the DDEML API described in the last column shields applications (and the OLE client and server libraries) from the messy details of DDE transactions and allows for future improvements to the DDE mechanisms that will be transparent to applications that use DDEML, the fact remains that DDE as we know it today is a shaky foundation for a protocol as complex as OLE.

OLE also offers users a chance to really paint themselves into a corner. If a user has many different OLE-aware applications on his disk and uses them all to build complex documents that are chockfull of many different types of embedded objects, he may find that he's the only person in the known universe with a software/hardware configuration that is capable of presenting, editing, and printing the document. If he passes the document off to another user who doesn't have a system with the exact same suite of applications in the exact same versions, that user will just see a document with a hole that represents each object for which there is no server.

If you'd like more information about OLE, you should browse through the Object Linking and Embedding Extensible Application Protocol document, which is available from Microsoft Corp., One Microsoft Way, Redmond, WA 98052. I referred to the "January 1991 Beta Version" while writing this column, but an updated specification will undoubtedly be available by the time you read this. Prototype OLE-aware applications (Cardfile, Paintbrush, Shapes) and example source code can be downloaded from the Microsoft Forum on CompuServe.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:

PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■